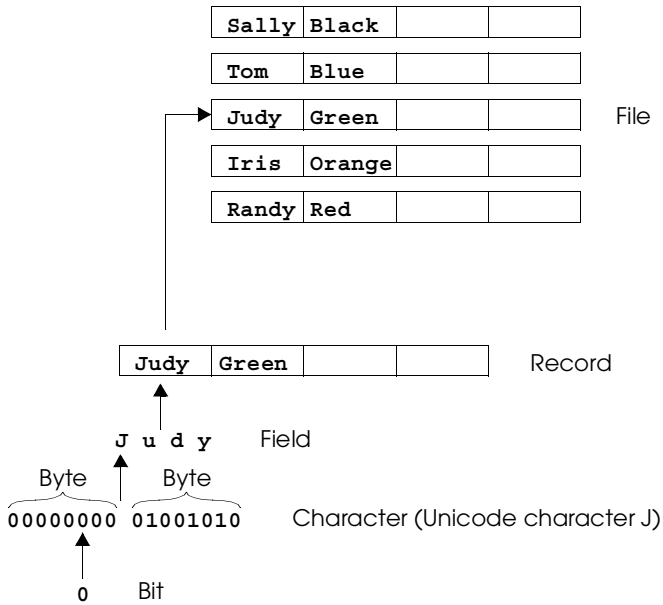

Fig. 17.1 The data hierarchy.

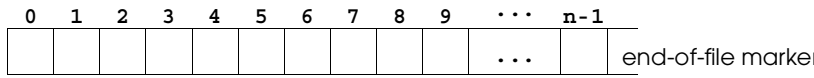

Fig. 17.2 Java's view of a file of ***n*** bytes.

A portion of the class hierarchy of the `java.io` package

```
java.lang.Object
   File
   FileDescriptor
   InputStream
      ByteArrayInputStream
      FileInputStream
      FilterInputStream
         BufferedInputStream
         DataInputStream
         PushbackInputStream
      ObjectInputStream
      PipedInputStream
      SequenceInputStream
   OutputStream
      ByteArrayOutputStream
      FileOutputStream
      FilterOutputStream
         BufferedOutputStream
         DataOutputStream
         PrintStream
      ObjectOutputStream
      PipedOutputStream
   RandomAccessFile
   Reader
      BufferedReader
         LineNumberReader
      CharArrayReader
      FilterReader
         PushbackReader
      InputStreamReader
         FileReader
      PipedReader
      StringReader
   Writer
```

Fig. 17.3 A portion of the class hierarchy of the **`java.io`** package (part 1 of 2).

A portion of the class hierarchy of the `java.io` package

```
BufferedWriter
CharArrayWriter
FilterWriter

OutputStreamWriter
   FileWriter
PipedWriter
PrintWriter
StringWriter
```

Fig. 17.3 A portion of the class hierarchy of the **`java.io`** package (part 2 of 2).

```
1  // Fig. 17.4: BankUI.java
2  // A reusable GUI for the examples in this chapter.
3  package com.deitel.jhtp3.ch17;
4  import java.awt.*;
5  import javax.swing.*;
6
7  public class BankUI extends JPanel {
8     protected final static String names[] = { "Account number",
9        "First name", "Last name", "Balance",
10       "Transaction Amount" };
11    protected JLabel labels[];
12    protected JTextField fields[];
13    protected JButton doTask, doTask2;
14    protected JPanel innerPanelCenter, innerPanelSouth;
15    protected int size = 4;
16    public static final int ACCOUNT = 0, FIRST = 1, LAST = 2,
17                            BALANCE = 3, TRANSACTION = 4;
18
19    public BankUI()
20    {
21       this( 4 );
22    }
23
24    public BankUI( int mySize )
25    {
26       size = mySize;
27       labels = new JLabel[ size ];
28       fields = new JTextField[ size ];
29
30       for ( int i = 0; i < labels.length; i++ )
31          labels[ i ] = new JLabel( names[ i ] );
32
33       for ( int i = 0; i < fields.length; i++ )
34          fields[ i ] = new JTextField();
35
36       innerPanelCenter = new JPanel();
37       innerPanelCenter.setLayout( new GridLayout( size, 2 ) );
38
39       for ( int i = 0; i < size; i++ ) {
40          innerPanelCenter.add( labels[ i ] );
41          innerPanelCenter.add( fields[ i ] );
42       }
43
```

Fig. 17.4 Creating a sequential file (part 1 of 7).

```
44          doTask = new JButton();
45          doTask2 = new JButton();
46          innerPanelSouth = new JPanel();
47          innerPanelSouth.add( doTask2 );
48          innerPanelSouth.add( doTask );
49
50          setLayout( new BorderLayout() );
51          add( innerPanelCenter, BorderLayout.CENTER );
52          add( innerPanelSouth, BorderLayout.SOUTH );
53          validate();
54       }
55
56       public JButton getDoTask() { return doTask; }
57
58       public JButton getDoTask2() { return doTask2; }
59
60       public JTextField[] getFields() { return fields; }
61
62       public void clearFields()
63       {
64          for ( int i = 0; i < size; i++ )
65             fields[ i ].setText( "" );
66       }
67
68       public void setFieldValues( String s[] )
69          throws IllegalArgumentException
70       {
71          if ( s.length != size )
72             throw new IllegalArgumentException( "There must be "
73                + size + " Strings in the array" );
74
75          for ( int i = 0; i < size; i++ )
76             fields[ i ].setText( s[ i ] );
77       }
78
79       public String[] getFieldValues()
80       {
81          String values[] = new String[ size ];
82
83          for ( int i = 0; i < size; i++ )
84             values[ i ] = fields[ i ].getText();
85
86          return values;
87       }
88    }
```

Fig. 17.4 Creating a sequential file (part 2 of 7).

```
89    // Fig. 17.4: BankAccountRecord.java
90    // A class that represents one record of information.
91    package com.deitel.jhtp3.ch17;
92    import java.io.Serializable;
93
```

```
94  public class BankAccountRecord implements Serializable {
95     private int account;
96     private String firstName;
97     private String lastName;
98     private double balance;
99
100    public BankAccountRecord()
101    {
102       this( 0, "", "", 0.0 );
103    }
104
105    public BankAccountRecord( int acct, String first,
106                              String last, double bal )
107    {
108       setAccount( acct );
109       setFirstName( first );
110       setLastName( last );
111       setBalance( bal );
112    }
113
114    public void setAccount( int acct )
115    {
116       account = acct;
117    }
118
119    public int getAccount() { return account; }
120
121    public void setFirstName( String first )
122    {
123       firstName = first;
124    }
125
126    public String getFirstName() { return firstName; }
127
128    public void setLastName( String last )
129    {
130       lastName = last;
131    }
132
133    public String getLastName() { return lastName; }
134
135    public void setBalance( double bal )
136    {
137       balance = bal;
138    }
139
140    public double getBalance() { return balance; }
141 }
```

Fig. 17.4 Creating a sequential file (part 3 of 7).

```
142 // Fig. 17.4: CreateSequentialFile.java
143 // Demonstrating object output with class ObjectOutputStream.
144 // The objects are written sequentially to a file.
```

```
145 import java.io.*;
146 import java.awt.*;
147 import java.awt.event.*;
148 import javax.swing.*;
149 import com.deitel.jhtp3.ch17.BankUI;
150 import com.deitel.jhtp3.ch17.BankAccountRecord;
151
152 public class CreateSequentialFile extends JFrame {
153    private ObjectOutputStream output;
154    private BankUI userInterface;
155    private JButton enter, open;
156
157    public CreateSequentialFile()
158    {
159       super( "Creating a Sequential File of Objects" );
160
161       getContentPane().setLayout( new BorderLayout() );
162       userInterface = new BankUI();
163
164       enter = userInterface.getDoTask();
165       enter.setText( "Enter" );
166       enter.setEnabled( false );  // disable button to start
167       enter.addActionListener(
168          new ActionListener() {
169             public void actionPerformed( ActionEvent e )
170             {
171                addRecord();
172             }
173          }
174       );
175
176       addWindowListener(
177          new WindowAdapter() {
178             public void windowClosing( WindowEvent e )
179             {
180                if ( output != null ) {
181                   addRecord();
182                   closeFile();
183                }
184                else
185                   System.exit( 0 );
186             }
187          }
188       );
189       open = userInterface.getDoTask2();
190
191       open.setText( "Save As" );
192       open.addActionListener(
193          new ActionListener() {
```

Fig. 17.4 Creating a sequential file (part 4 of 7).

```
194             public void actionPerformed( ActionEvent e )
195             {
196                openFile();
```

```
            }
         }
      );
      getContentPane().add( userInterface,
                            BorderLayout.CENTER );

      setSize( 300, 200 );
      show();
   }

   private void openFile()
   {
      JFileChooser fileChooser = new JFileChooser();
      fileChooser.setFileSelectionMode(
         JFileChooser.FILES_ONLY );

      int result = fileChooser.showSaveDialog( this );

      // user clicked Cancel button on dialog
      if ( result == JFileChooser.CANCEL_OPTION )
         return;

      File fileName = fileChooser.getSelectedFile();

      if ( fileName == null ||
           fileName.getName().equals( "" ) )
         JOptionPane.showMessageDialog( this,
            "Invalid File Name",
            "Invalid File Name",
            JOptionPane.ERROR_MESSAGE );
      else {
         // Open the file
         try {
            output = new ObjectOutputStream(
                        new FileOutputStream( fileName ) );
            open.setEnabled( false );
            enter.setEnabled( true );
         }
         catch ( IOException e ) {
            JOptionPane.showMessageDialog( this,
               "Error Opening File", "Error",
               JOptionPane.ERROR_MESSAGE );
         }
      }
   }

   private void closeFile()
   {
      try {
         output.close();
```

Fig. 17.4 Creating a sequential file (part 5 of 7).

```
         System.exit( 0 );
```

```
248         }
249         catch( IOException ex ) {
250            JOptionPane.showMessageDialog( this,
251               "Error closing file",
252               "Error", JOptionPane.ERROR_MESSAGE );
253            System.exit( 1 );
254         }
255      }
256
257      public void addRecord()
258      {
259         int accountNumber = 0;
260         BankAccountRecord record;
261         String fieldValues[] = userInterface.getFieldValues();
262
263         // If the account field value is not empty
264         if ( ! fieldValues[ 0 ].equals( "" ) ) {
265            // output the values to the file
266            try {
267               accountNumber =
268                  Integer.parseInt( fieldValues[ 0 ] );
269
270               if ( accountNumber > 0 ) {
271                  record = new BankAccountRecord(
272                     accountNumber, fieldValues[ 1 ],
273                     fieldValues[ 2 ],
274                     Double.parseDouble( fieldValues[ 3 ] ) );
275                  output.writeObject( record );
276                  output.flush();
277               }
278
279               // clear the TextFields
280               userInterface.clearFields();
281            }
282            catch ( NumberFormatException nfe ) {
283               JOptionPane.showMessageDialog( this,
284                  "Bad account number or balance",
285                  "Invalid Number Format",
286                  JOptionPane.ERROR_MESSAGE );
287            }
288            catch ( IOException io ) {
289               closeFile();
290            }
291         }
292      }
293
294      public static void main( String args[] )
295      {
296         new CreateSequentialFile();
297      }
298   }
```

Fig. 17.4 Creating a sequential file (part 6 of 7).

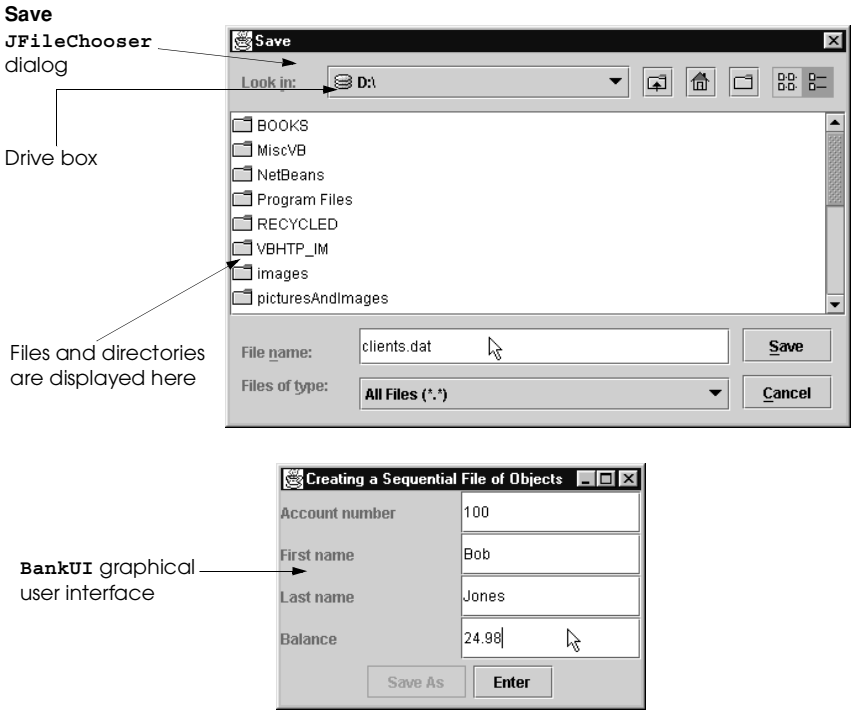

Fig. 17.4 Creating a sequential file (part 7 of 7).

.

Sample Data			
100	Bob	Jones	24.98
200	Steve	Doe	-345.67
300	Pam	White	0.00
400	Sam	Stone	-42.16
500	Sue	Rich	224.62

Fig. 17.5 Sample data for the program of Fig. 17.4.

```
1  // Fig. 17.6: ReadSequentialFile.java
2  // This program reads a file of objects sequentially
3  // and displays each record.
4  import java.io.*;
5  import java.awt.*;
6  import java.awt.event.*;
7  import javax.swing.*;
8  import com.deitel.jhtp3.ch17.*;
9
10 public class ReadSequentialFile extends JFrame {
11    private ObjectInputStream input;
12    private BankUI userInterface;
13    private JButton nextRecord, open;
14
15    // Constructor -- initialize the Frame
16    public ReadSequentialFile()
17    {
18       super( "Reading a Sequential File of Objects" );
19
20       getContentPane().setLayout( new BorderLayout() );
21       userInterface = new BankUI();
22       nextRecord = userInterface.getDoTask();
23       nextRecord.setText( "Next Record" );
24       nextRecord.setEnabled( false );
25
26       nextRecord.addActionListener(
27          new ActionListener() {
28             public void actionPerformed( ActionEvent e )
29             {
30                readRecord();
31             }
32          }
33       );
34
35       addWindowListener(
36          new WindowAdapter() {
37             public void windowClosing( WindowEvent e )
38             {
39                if ( input != null )
40                   closeFile();
41
```

Fig. 17.6 Reading a sequential file (part 1 of 4).

```
42                System.exit( 0 );
43             }
44          }
45       );
46       open = userInterface.getDoTask2();
47
48       open.setText( "Open File" );
49       open.addActionListener(
50          new ActionListener() {
```

```
            public void actionPerformed( ActionEvent e )
            {
               openFile();
            }
         }
      );

      getContentPane().add( userInterface,
                            BorderLayout.CENTER );
      pack();
      setSize( 300, 200 );
      show();
   }

   private void openFile()
   {
      JFileChooser fileChooser = new JFileChooser();

      fileChooser.setFileSelectionMode(
         JFileChooser.FILES_ONLY );
      int result = fileChooser.showOpenDialog( this );

      // user clicked Cancel button on dialog
      if ( result == JFileChooser.CANCEL_OPTION )
         return;

      File fileName = fileChooser.getSelectedFile();

      if ( fileName == null ||
           fileName.getName().equals( "" ) )
         JOptionPane.showMessageDialog( this,
            "Invalid File Name",
            "Invalid File Name",
            JOptionPane.ERROR_MESSAGE );
      else {
         // Open the file
         try {
            input = new ObjectInputStream(
                        new FileInputStream( fileName ) );
            open.setEnabled( false );
            nextRecord.setEnabled( true );
         }
```

Fig. 17.6 Reading a sequential file (part 2 of 4).

```
         catch ( IOException e ) {
            JOptionPane.showMessageDialog( this,
               "Error Opening File", "Error",
               JOptionPane.ERROR_MESSAGE );
         }
      }
   }

   public void readRecord()
   {
```

```
103         BankAccountRecord record;
104
105         // input the values from the file
106         try {
107            record = ( BankAccountRecord ) input.readObject();
108            String values[] = {
109               String.valueOf( record.getAccount() ),
110               record.getFirstName(),
111               record.getLastName(),
112               String.valueOf( record.getBalance() ) };
113            userInterface.setFieldValues( values );
114         }
115         catch ( EOFException eofex ) {
116            nextRecord.setEnabled( false );
117            JOptionPane.showMessageDialog( this,
118               "No more records in file",
119               "End of File", JOptionPane.ERROR_MESSAGE );
120         }
121         catch ( ClassNotFoundException cnfex ) {
122            JOptionPane.showMessageDialog( this,
123               "Unable to create object",
124               "Class Not Found", JOptionPane.ERROR_MESSAGE );
125         }
126         catch ( IOException ioex ) {
127            JOptionPane.showMessageDialog( this,
128               "Error during read from file",
129               "Read Error", JOptionPane.ERROR_MESSAGE );
130         }
131      }
132
133      private void closeFile()
134      {
135         try {
136            input.close();
137            System.exit( 0 );
138         }
139         catch ( IOException e ) {
140            JOptionPane.showMessageDialog( this,
141               "Error closing file",
142               "Error", JOptionPane.ERROR_MESSAGE );
143            System.exit( 1 );
144         }
145      }
```

Fig. 17.6 Reading a sequential file (part 3 of 4).

```
146
147      public static void main( String args[] )
148      {
149         new ReadSequentialFile();
150      }
151   }
```

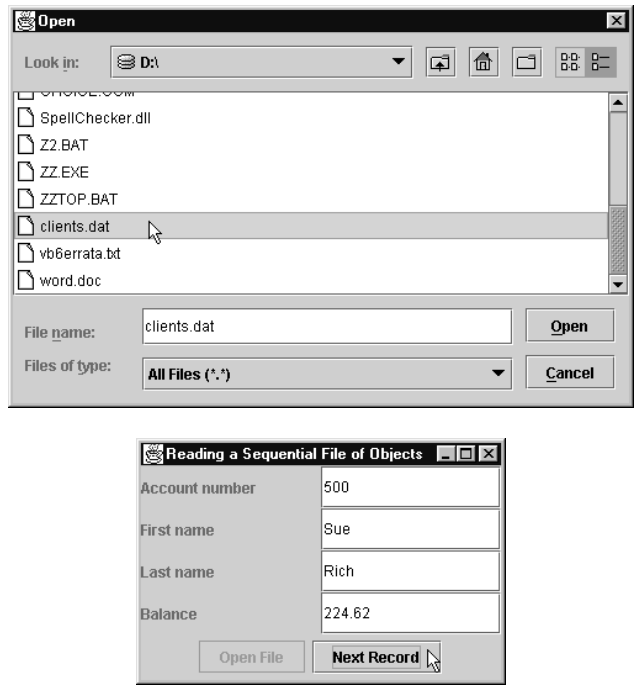

Fig. 17.6 Reading a sequential file (part 4 of 4).

```
1  // Fig. 17.7: CreditInquiry.java
2  // This program reads a file sequentially and displays the
3  // contents in a text area based on the type of account the
4  // user requests (credit balance, debit balance or
5  // zero balance).
6  import java.io.*;
7  import java.awt.*;
8  import java.awt.event.*;
9  import java.text.DecimalFormat;
10 import javax.swing.*;
11 import com.deitel.jhtp3.ch17.BankAccountRecord;
12
13 public class CreditInquiry extends JFrame {
14    private JTextArea recordDisplay;
15    private JButton open, done, credit, debit, zero;
16    private JPanel buttonPanel;
17    private ObjectInputStream input;
18    private FileInputStream fileInput;
19    private File fileName;
20    private String accountType;
21
```

Fig. 17.7 Credit inquiry program (part 1 of 5).

```
22    public CreditInquiry()
23    {
24       super( "Credit Inquiry Program" );
25
26       Container c = getContentPane();
27       c.setLayout( new BorderLayout() );
28       buttonPanel = new JPanel();
29
30       open = new JButton( "Open File" );
31       open.addActionListener(
32          new ActionListener() {
33             public void actionPerformed( ActionEvent e )
34             {
35                openFile( true );
36             }
37          }
38       );
39       buttonPanel.add( open );
40
41       credit = new JButton( "Credit balances" );
42       credit.addActionListener(
43          new ActionListener() {
44             public void actionPerformed( ActionEvent e )
45             {
46                accountType = e.getActionCommand();
47                readRecords();
48             }
49          }
50       );
```

```
51          buttonPanel.add( credit );
52
53          debit = new JButton( "Debit balances" );
54          debit.addActionListener(
55             new ActionListener() {
56                public void actionPerformed( ActionEvent e )
57                {
58                   accountType = e.getActionCommand();
59                   readRecords();
60                }
61             }
62          );
63          buttonPanel.add( debit );
64
65          zero = new JButton( "Zero balances" );
66          zero.addActionListener(
67             new ActionListener() {
68                public void actionPerformed( ActionEvent e )
69                {
70                   accountType = e.getActionCommand();
71                   readRecords();
72                }
73             }
74          );
```

Fig. 17.7 Credit inquiry program (part 2 of 5).

```
75          buttonPanel.add( zero );
76
77          done = new JButton( "Done" );
78          buttonPanel.add( done );
79          done.addActionListener(
80             new ActionListener() {
81                public void actionPerformed( ActionEvent e )
82                {
83                   if ( fileInput != null )
84                      closeFile();
85
86                   System.exit( 0 );
87                }
88             }
89          );
90
91          recordDisplay = new JTextArea();
92       JScrollPane scroller = new JScrollPane( recordDisplay );
93          c.add( scroller, BorderLayout.CENTER );
94          c.add( buttonPanel, BorderLayout.SOUTH );
95
96          credit.setEnabled( false );
97          debit.setEnabled( false );
98          zero.setEnabled( false );
99
100         pack();
101         setSize( 600, 250 );
102         show();
```

```
103    }
104
105    private void openFile( boolean firstTime )
106    {
107       if ( firstTime ) {
108          JFileChooser fileChooser = new JFileChooser();
109
110          fileChooser.setFileSelectionMode(
111             JFileChooser.FILES_ONLY );
112          int result = fileChooser.showOpenDialog( this );
113
114          // user clicked Cancel button on dialog
115          if ( result == JFileChooser.CANCEL_OPTION )
116             return;
117
118          fileName = fileChooser.getSelectedFile();
119       }
120
121       if ( fileName == null ||
122            fileName.getName().equals( "" ) )
123          JOptionPane.showMessageDialog( this,
124             "Invalid File Name",
125             "Invalid File Name",
126             JOptionPane.ERROR_MESSAGE );
```

Fig. 17.7 Credit inquiry program (part 3 of 5).

```
127       else {
128          // Open the file
129          try {
130             // close file from previous operation
131             if ( input != null )
132                input.close();
133
134             fileInput = new FileInputStream( fileName );
135             input = new ObjectInputStream( fileInput );
136             open.setEnabled( false );
137             credit.setEnabled( true );
138             debit.setEnabled( true );
139             zero.setEnabled( true );
140          }
141          catch ( IOException e ) {
142             JOptionPane.showMessageDialog( this,
143                "File does not exist",
144                "Invalid File Name",
145                JOptionPane.ERROR_MESSAGE );
146          }
147       }
148    }
149
150    private void closeFile()
151    {
152       try {
153          input.close();
154       }
```

```
155         catch ( IOException ioe ) {
156            JOptionPane.showMessageDialog( this,
157               "Error closing file",
158               "Error", JOptionPane.ERROR_MESSAGE );
159            System.exit( 1 );
160         }
161      }
162
163      private void readRecords()
164      {
165         BankAccountRecord record;
166         DecimalFormat twoDigits = new DecimalFormat( "0.00" );
167         openFile( false );
168
169         try {
170            recordDisplay.setText( "The accounts are:\n" );
171
172            // input the values from the file
173            while ( true ) {
174               record =
175                  ( BankAccountRecord ) input.readObject();
176
```

Fig. 17.7 Credit inquiry program (part 4 of 5).

```
177               if ( shouldDisplay( record.getBalance() ) )
178                  recordDisplay.append( record.getAccount() +
179                     "\t" + record.getFirstName() + "\t" +
180                     record.getLastName() + "\t" +
181                     twoDigits.format( record.getBalance() ) +
182                     "\n" );
183            }
184         }
185         catch ( EOFException eof ) {
186            closeFile();
187         }
188         catch ( ClassNotFoundException cnfex ) {
189            JOptionPane.showMessageDialog( this,
190               "Unable to create object",
191               "Class Not Found", JOptionPane.ERROR_MESSAGE );
192         }
193         catch ( IOException e ) {
194            JOptionPane.showMessageDialog( this,
195               "Error reading from file",
196               "Error", JOptionPane.ERROR_MESSAGE );
197         }
198      }
199
200      private boolean shouldDisplay( double balance )
201      {
202         if ( accountType.equals( "Credit balances" ) &&
203              balance < 0 )
204            return true;
205         else if ( accountType.equals( "Debit balances" ) &&
206                   balance > 0 )
```

```
207            return true;
208         else if ( accountType.equals( "Zero balances" ) &&
209                   balance == 0 )
210            return true;
211
212         return false;
213      }
214
215      public static void main( String args[] )
216      {
217         final CreditInquiry app = new CreditInquiry();
218
219         app.addWindowListener(
220            new WindowAdapter() {
221               public void windowClosing( WindowEvent e )
222               {
223                  app.closeFile();
224                  System.exit( 0 );
225               }
226            }
227         );
228      }
229   }
```

Fig. 17.7 Credit inquiry program (part 5 of 5).

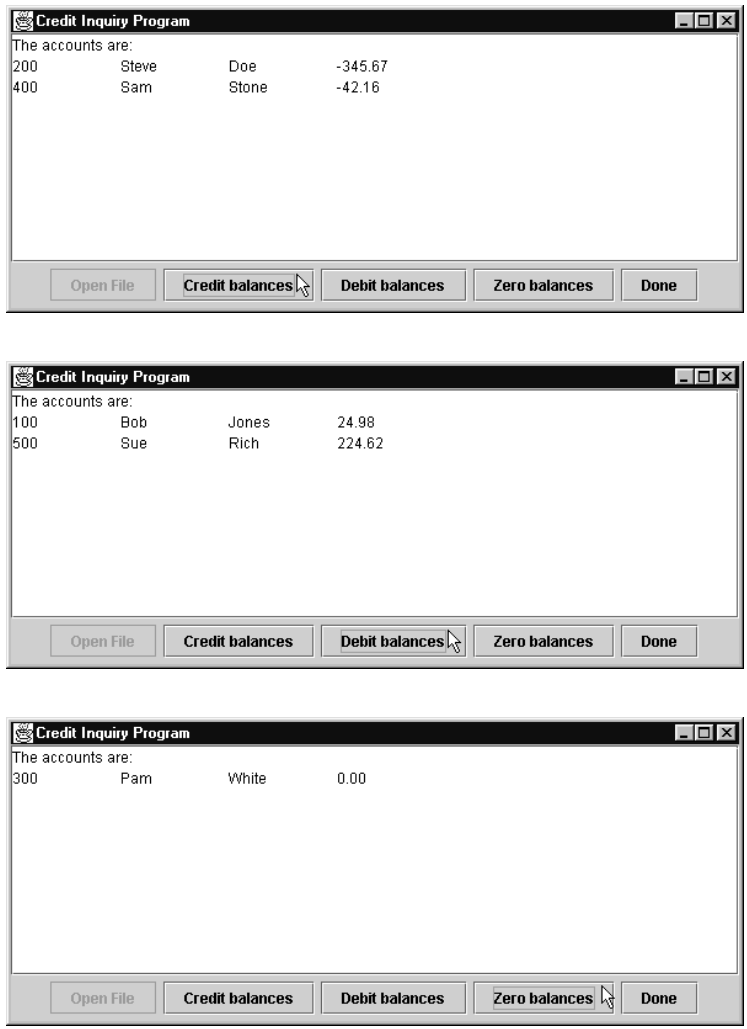

Fig. 17.8 Sample outputs of the credit inquiry program of Fig. 17.7.

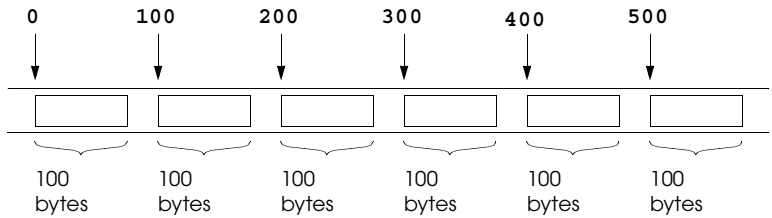

Fig. 17.9 Java's view of a random-access file.

```
1  // Fig. 17.10: Record.java
2  // Record class for the RandomAccessFile programs.
3  package com.deitel.jhtp3.ch17;
4  import java.io.*;
5  import com.deitel.jhtp3.ch17.BankAccountRecord;
6
7  public class Record extends BankAccountRecord {
8
9     public Record()
10    {
11       this( 0, "", "", 0.0 );
12    }
13
14    public Record( int acct, String first,
15                   String last, double bal )
16    {
```

Fig. 17.10 `Record` class used in the random-access file programs (part 1 of 2).

```
17       super( acct, first, last, bal );
18    }
19
20    // Read a record from the specified RandomAccessFile
21    public void read( RandomAccessFile file ) throws IOException
22    {
23       setAccount( file.readInt() );
24       setFirstName( padName( file ) );
25       setLastName( padName( file ) );
26       setBalance( file.readDouble() );
27    }
28
29    private String padName( RandomAccessFile f )
30       throws IOException
31    {
32       char name[] = new char[ 15 ], temp;
33
34       for ( int i = 0; i < name.length; i++ ) {
35          temp = f.readChar();
36          name[ i ] = temp;
37       }
38
39       return new String( name ).replace( '\0', ' ' );
40    }
41
42    // Write a record to the specified RandomAccessFile
43    public void write( RandomAccessFile file ) throws IOException
44    {
45       file.writeInt( getAccount() );
46       writeName( file, getFirstName() );
47       writeName( file, getLastName() );
48       file.writeDouble( getBalance() );
49    }
50
51    private void writeName( RandomAccessFile f, String name )
52       throws IOException
```

```
53     {
54        StringBuffer buf = null;
55
56        if ( name != null )
57           buf = new StringBuffer( name );
58        else
59           buf = new StringBuffer( 15 );
60
61        buf.setLength( 15 );
62        f.writeChars( buf.toString() );
63     }
64
65     // NOTE: This method contains a hard coded value for the
66     // size of a record of information.
67     public static int size() { return 72; }
68  }
```

Fig. 17.10 **`Record`** class used in the random-access file programs (part 2 of 2).

```
1  // Fig. 17.11: CreateRandFile.java
2  // This program creates a random access file sequentially
3  // by writing 100 empty records to disk.
4  import com.deitel.jhtp3.ch17.Record;
5  import java.io.*;
6  import javax.swing.*;
7
8  public class CreateRandomFile {
9     private Record blank;
10    private RandomAccessFile file;
11
12    public CreateRandomFile()
13    {
14       blank = new Record();
15       openFile();
16    }
```

Fig. 17.11 Creating a random-access file sequentially (part 1 of 2).

```
17
18    private void openFile()
19    {
20       JFileChooser fileChooser = new JFileChooser();
21                    fileChooser.setFileSelectionMode(
22                                   JFileChooser.FILES_ONLY );
23       int result = fileChooser.showSaveDialog( null );
24
25       // user clicked Cancel button on dialog
26       if ( result == JFileChooser.CANCEL_OPTION )
27          return;
28
29       File fileName = fileChooser.getSelectedFile();
30
31       if ( fileName == null ||
32            fileName.getName().equals( "" ) )
33          JOptionPane.showMessageDialog( null,
34             "Invalid File Name",
35             "Invalid File Name",
36             JOptionPane.ERROR_MESSAGE );
37       else {
38          // Open the file
39          try {
40             file = new RandomAccessFile( fileName, "rw" );
41
42             for ( int i = 0; i < 100; i++ )
43                blank.write( file );
44
45             System.exit( 0 );
46          }
47          catch ( IOException e ) {
48             JOptionPane.showMessageDialog( null,
49                "File does not exist",
50                "Invalid File Name",
```

```
51                JOptionPane.ERROR_MESSAGE );
52             System.exit( 1 );
53          }
54       }
55    }
56
57    public static void main( String args[] )
58    {
59       new CreateRandomFile();
60    }
61 }
```

Fig. 17.11 Creating a random-access file sequentially (part 2 of 2).

```
1  // Fig. 17.12: WriteRandomFile.java
2  // This program uses TextFields to get information from the
3  // user at the keyboard and writes the information to a
4  // random-access file.
5  import com.deitel.jhtp3.ch17.*;
6  import javax.swing.*;
7  import java.io.*;
8  import java.awt.event.*;
9  import java.awt.*;
10
11 public class WriteRandomFile extends JFrame {
12    private RandomAccessFile output;
13    private BankUI userInterface;
14    private JButton enter, open;
15
16    // Constructor -- intialize the Frame
17    public WriteRandomFile()
18    {
19       super( "Write to random access file" );
20
21       userInterface = new BankUI();
22       enter = userInterface.getDoTask();
23       enter.setText( "Enter" );
24       enter.setEnabled( false );
25       enter.addActionListener(
26          new ActionListener() {
27             public void actionPerformed( ActionEvent e )
28             {
29                addRecord();
30             }
31          }
32       );
```

Fig. 17.12 Writing data randomly to a random-access file (part 1 of 4).

```
33
34       addWindowListener(
35          new WindowAdapter() {
36             public void windowClosing( WindowEvent e )
37             {
38                if ( output != null ) {
39                   addRecord();
40                   closeFile();
41                }
42                else
43                   System.exit( 0 );
44             }
45          }
46       );
47       open = userInterface.getDoTask2();
48
49       open.setText( "Save As" );
50       open.addActionListener(
```

```
51             new ActionListener() {
52                public void actionPerformed( ActionEvent e )
53                {
54                   // Open the file
55                   openFile();
56                }
57             }
58          );
59          getContentPane().add( userInterface,
60                                BorderLayout.CENTER );
61
62          setSize( 300, 150 );
63          show();
64       }
65
66       private void openFile()
67       {
68          JFileChooser fileChooser = new JFileChooser();
69
70          fileChooser.setFileSelectionMode(
71             JFileChooser.FILES_ONLY );
72          int result = fileChooser.showSaveDialog( this );
73
74          // user clicked Cancel button on dialog
75          if ( result == JFileChooser.CANCEL_OPTION )
76             return;
77
78          File fileName = fileChooser.getSelectedFile();
79
80          if ( fileName == null ||
81               fileName.getName().equals( "" ) )
82             JOptionPane.showMessageDialog( this,
83                "Invalid File Name",
84                "Invalid File Name",
85                JOptionPane.ERROR_MESSAGE );
```

Fig. 17.12 Writing data randomly to a random-access file (part 2 of 4).

```
86          else {
87             // Open the file
88             try {
89                output = new RandomAccessFile( fileName, "rw" );
90                enter.setEnabled( true );
91                open.setEnabled( false );
92             }
93             catch ( IOException e ) {
94                JOptionPane.showMessageDialog( this,
95                   "File does not exist",
96                   "Invalid File Name",
97                   JOptionPane.ERROR_MESSAGE );
98             }
99          }
100      }
101
```

```
102    private void closeFile()
103    {
104       try {
105          output.close();
106          System.exit( 0 );
107       }
108       catch( IOException ex ) {
109          JOptionPane.showMessageDialog( this,
110             "Error closing file",
111             "Error", JOptionPane.ERROR_MESSAGE );
112          System.exit( 1 );
113       }
114    }
115
116    public void addRecord()
117    {
118       int accountNumber = 0;
119       String fields[] = userInterface.getFieldValues();
120       Record record = new Record();
121
122       if ( !fields[ BankUI.ACCOUNT ].equals( "" ) ) {
123          // output the values to the file
124          try {
125             accountNumber =
126                Integer.parseInt( fields[ BankUI.ACCOUNT ] );
127
128             if ( accountNumber > 0 && accountNumber <= 100 ) {
129                record.setAccount( accountNumber );
130
131                record.setFirstName( fields[ BankUI.FIRST ] );
132                record.setLastName( fields[ BankUI.LAST ] );
133                record.setBalance( Double.parseDouble(
134                                   fields[ BankUI.BALANCE ] ) );
135
136                output.seek( ( accountNumber - 1 ) *
137                             Record.size() );
```

Fig. 17.12 Writing data randomly to a random-access file (part 3 of 4).

```
138                record.write( output );
139             }
140
141             userInterface.clearFields();  // clear TextFields
142          }
143          catch ( NumberFormatException nfe ) {
144                JOptionPane.showMessageDialog( this,
145                   "Bad account number or balance",
146                   "Invalid Number Format",
147                   JOptionPane.ERROR_MESSAGE );
148          }
149          catch ( IOException io ) {
150             closeFile();
151          }
152       }
153    }
```

```
154
155     // Create a WriteRandomFile object and start the program
156     public static void main( String args[] )
157     {
158        new WriteRandomFile();
159     }
160  }
```

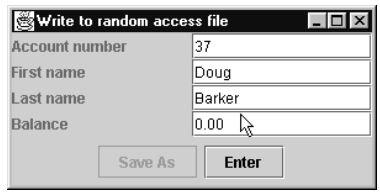

Fig. 17.12 Writing data randomly to a random-access file (part 4 of 4).

```
// Fig. 17.13: ReadRandomFile.java
// This program reads a random-access file sequentially and
// displays the contents one record at a time in text fields.
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import java.text.DecimalFormat;
import javax.swing.*;
import com.deitel.jhtp3.ch17.*;

public class ReadRandomFile extends JFrame {
   private BankUI userInterface;
   private RandomAccessFile input;
   private JButton next, open;

   public ReadRandomFile()
   {
      super( "Read Client File" );

      userInterface = new BankUI();
      next = userInterface.getDoTask();
      next.setText( "Next" );
      next.setEnabled( false );
      next.addActionListener(
         new ActionListener() {
            public void actionPerformed( ActionEvent e )
            {
               readRecord();
            }
         }
      );

      addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
```

Fig. 17.13 Reading a random-access file sequentially (part 1 of 4).

```
               if ( input != null ) {
                  closeFile();
               }
               else
                  System.exit( 0 );
            }
         }
      );
      open = userInterface.getDoTask2();

      open.setText( "Read File" );
      open.addActionListener(
         new ActionListener() {
            public void actionPerformed( ActionEvent e )
```

```
            {
               openFile();
            }
         }
      );
      getContentPane().add( userInterface );

      setSize( 300, 150 );
      show();
   }

   private void openFile()
   {
      JFileChooser fileChooser = new JFileChooser();

      fileChooser.setFileSelectionMode(
         JFileChooser.FILES_ONLY );
      int result = fileChooser.showOpenDialog( this );

      // user clicked Cancel button on dialog
      if ( result == JFileChooser.CANCEL_OPTION )
         return;

      File fileName = fileChooser.getSelectedFile();

      if ( fileName == null ||
           fileName.getName().equals( "" ) )
         JOptionPane.showMessageDialog( this,
            "Invalid File Name",
            "Invalid File Name",
            JOptionPane.ERROR_MESSAGE );
      else {
         // Open the file
         try {
            input = new RandomAccessFile( fileName, "r" );
            next.setEnabled( true );
            open.setEnabled( false );
         }
```

Fig. 17.13 Reading a random-access file sequentially (part 2 of 4).

```
         catch ( IOException e ) {
            JOptionPane.showMessageDialog( this,
               "File does not exist",
               "Invalid File Name",
               JOptionPane.ERROR_MESSAGE );
         }
      }
   }

   public void readRecord()
   {
      DecimalFormat twoDigits = new DecimalFormat( "0.00" );
      Record record = new Record();

```

```
103         // read a record and display
104         try {
105            do {
106               record.read( input );
107            } while ( record.getAccount() == 0 );
108
109            String values[] = {
110               String.valueOf( record.getAccount() ),
111               record.getFirstName(),
112               record.getLastName(),
113               String.valueOf( record.getBalance() ) };
114            userInterface.setFieldValues( values );
115         }
116         catch ( EOFException eof ) {
117            closeFile();
118         }
119         catch ( IOException e ) {
120            JOptionPane.showMessageDialog( this,
121               "Error Reading File",
122               "Error",
123               JOptionPane.ERROR_MESSAGE );
124            System.exit( 1 );
125         }
126      }
127
128      private void closeFile()
129      {
130         try {
131            input.close();
132            System.exit( 0 );
133         }
134         catch( IOException ex ) {
135            JOptionPane.showMessageDialog( this,
136               "Error closing file",
137               "Error", JOptionPane.ERROR_MESSAGE );
138            System.exit( 1 );
139         }
140      }
```

Fig. 17.13 Reading a random-access file sequentially (part 3 of 4).

```
141
142      public static void main( String args[] )
143      {
144         new ReadRandomFile();
145      }
146   }
```

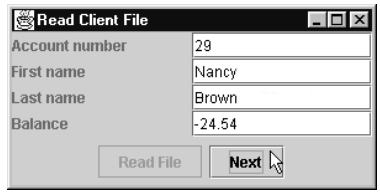

Fig. 17.13 Reading a random-access file sequentially (part 4 of 4).

```
1  // Fig. 17.14: TransactionProcessor.java
2  // Transaction processing program using RandomAccessFiles.
3  // This program reads a random-access file sequentially,
4  // updates record already written to the file, creates new
```

Fig. 17.14 Transaction-processing program (part 1 of 12).

```
5  // record to be placed in the file and deletes data
6  // already in the file.
7  import java.awt.*;
8  import java.awt.event.*;
9  import java.io.*;
10 import java.text.DecimalFormat;
11 import com.deitel.jhtp3.ch17.*;
12 import javax.swing.*;
13
14 public class TransactionProcessor extends JFrame  {
15    private JDesktopPane desktop;
16    private JButton open, updateRecord, newRecord, deleteRecord;
17    private JInternalFrame mainDialog;
18    private UpdateDialog updateDialog;
19    private NewDialog newDialog;
20    private DeleteDialog deleteDialog;
21    private RandomAccessFile file;
22    private Record record;
23
24    public TransactionProcessor()
25    {
26       super( "Transaction Processor" );
27
28       desktop = new JDesktopPane();
29
30       mainDialog = new JInternalFrame();
31       updateRecord = new JButton( "Update Record" );
32       updateRecord.setEnabled( false );
33       updateRecord.addActionListener(
34          new ActionListener() {
35             public void actionPerformed( ActionEvent e )
36             {
37                mainDialog.setVisible( false );
38                updateDialog.setVisible( true );
39             }
40          }
41       );
42
43       deleteRecord = new JButton( "Delete Record" );
44       deleteRecord.setEnabled( false );
45       deleteRecord.addActionListener(
46          new ActionListener() {
47             public void actionPerformed( ActionEvent e )
48             {
49                mainDialog.setVisible( false );
50                deleteDialog.setVisible( true );
```

```
51             }
52          }
53       );
54
55       newRecord = new JButton( "New Record" );
56       newRecord.setEnabled( false );
```

Fig. 17.14 Transaction-processing program (part 2 of 12).

```
57       newRecord.addActionListener(
58          new ActionListener() {
59             public void actionPerformed( ActionEvent e )
60             {
61                mainDialog.setVisible( false );
62                newDialog.setVisible( true );
63             }
64          }
65       );
66
67       open = new JButton( "New/Open File" );
68       open.addActionListener(
69          new ActionListener() {
70             public void actionPerformed( ActionEvent e )
71             {
72                open.setEnabled( false );
73                openFile();
74                ActionListener l = new ActionListener() {
75                   public void actionPerformed( ActionEvent e )
76
77                   {
78                     mainDialog.setVisible( true );
79                   }
80                };
81                updateDialog = new UpdateDialog( file, l );
82                desktop.add( updateDialog );
83                updateRecord.setEnabled( true );
84
85                deleteDialog = new DeleteDialog( file, l );
86                desktop.add ( deleteDialog );
87                deleteRecord.setEnabled( true );
88
89                newDialog = new NewDialog( file, l );
90                desktop.add( newDialog );
91                newRecord.setEnabled( true );
92
93             }
94          }
95       );
96
97       Container c = mainDialog.getContentPane();
98       c.setLayout( new GridLayout( 2, 2 ) );
99       c.add( updateRecord );
100      c.add( newRecord );
101      c.add( deleteRecord );
102      c.add( open );
```

```
103
104         setSize( 400, 250  );
105         mainDialog.setSize( 300, 80 );
106         desktop.add( mainDialog, BorderLayout.CENTER );
107         getContentPane().add( desktop );
```

Fig. 17.14 Transaction-processing program (part 3 of 12).

```
108         addWindowListener(
109            new WindowAdapter() {
110               public void windowClosing( WindowEvent e )
111               {
112                  if ( file != null )
113                     closeFile();
114
115                  System.exit( 0 );
116               }
117            }
118         );
119         show();
120      }
121
122      private void openFile()
123      {
124         JFileChooser fileChooser = new JFileChooser();
125
126         fileChooser.setFileSelectionMode(
127            JFileChooser.FILES_ONLY );
128
129         int result = fileChooser.showOpenDialog( this );
130
131         // user clicked Cancel button on dialog
132         if ( result == JFileChooser.CANCEL_OPTION )
133            return;
134
135         File fileName = fileChooser.getSelectedFile();
136
137         if ( fileName == null ||
138              fileName.getName().equals( "" ) )
139            JOptionPane.showMessageDialog( this,
140               "Invalid File Name",
141               "Invalid File Name",
142               JOptionPane.ERROR_MESSAGE );
143         else {
144            // Open the file
145            try {
146               file = new RandomAccessFile( fileName, "rw" );
147               updateRecord.setEnabled( true );
148               newRecord.setEnabled( true );
149               deleteRecord.setEnabled( true );
150               open.setEnabled( false );
151            }
152            catch ( IOException e ) {
153               JOptionPane.showMessageDialog( this,
154                  "File does not exist",
```

```
                "Invalid File Name",
                JOptionPane.ERROR_MESSAGE );
            }
         }
      }
```

Fig. 17.14 Transaction-processing program (part 4 of 12).

```

      private void closeFile()
      {
         try {
            file.close();
            System.exit( 0 );
         }
         catch( IOException ex ) {
            JOptionPane.showMessageDialog( this,
               "Error closing file",
               "Error", JOptionPane.ERROR_MESSAGE );
            System.exit( 1 );
         }
      }

      public static void main( String args[] )
      {
         new TransactionProcessor();
      }
   }

   class UpdateDialog extends JInternalFrame {
      private RandomAccessFile file;
      private BankUI userInterface;
      private JButton cancel, save;
      private JTextField account;

      public UpdateDialog( RandomAccessFile f, ActionListener l )
      {
         super( "Update Record" );

         file = f;
         userInterface = new BankUI( 5 );

         cancel = userInterface.getDoTask();
         cancel.setText( "Cancel" );
         cancel.addActionListener(
            new ActionListener() {
               public void actionPerformed( ActionEvent e )
               {
                  setVisible( false );
                  userInterface.clearFields();
               }
            }
         );
         cancel.addActionListener( l );

```

```
207         save = userInterface.getDoTask2();
208         save.setText( "Save Changes" );
209         save.addActionListener(
210            new ActionListener() {
211               public void actionPerformed( ActionEvent e )
212               {
```

Fig. 17.14 Transaction-processing program (part 5 of 12).

```
213                  addRecord( getRecord() );
214                  setVisible( false );
215                  userInterface.clearFields();
216               }
217            }
218         );
219         save.addActionListener( l );
220
221         JTextField transaction =
222            userInterface.getFields()[ BankUI.TRANSACTION ];
223         transaction.addActionListener(
224            new ActionListener() {
225               public void actionPerformed( ActionEvent e )
226               {
227                  try {
228                     Record record = getRecord();
229                     double change = Double.parseDouble(
230                              userInterface.getFieldValues()
231                              [ BankUI.TRANSACTION ] );
232                     String[] values = {
233                         String.valueOf( record.getAccount() ),
234                         record.getFirstName(),
235                         record.getLastName(),
236                         String.valueOf( record.getBalance()
237                                         + change ),
238                         "Charge(+) or payment (-)" };
239
240                      userInterface.setFieldValues( values );
241                  }
242                  catch ( NumberFormatException nfe ) {
243                    JOptionPane.showMessageDialog( new JFrame(),
244                         "Invalid Transaction",
245                         "Invalid Number Format",
246                         JOptionPane.ERROR_MESSAGE );
247                  }
248               }
249            }
250         );
251
252         account = userInterface.getFields()[ BankUI.ACCOUNT ];
253         account.addActionListener(
254            new ActionListener() {
255               public void actionPerformed( ActionEvent e )
256               {
257                  Record record = getRecord();
258
```

```
259                if ( record.getAccount() != 0 )  {
260                   String values[] = {
261                      String.valueOf( record.getAccount() ),
262                      record.getFirstName(),
263                      record.getLastName(),
264                      String.valueOf( record.getBalance() ),
265                      "Charge(+) or payment (-)" };
```

Fig. 17.14 Transaction-processing program (part 6 of 12).

```
266                      userInterface.setFieldValues( values );
267                   }
268                }
269             }
270       );
271       getContentPane().add( userInterface,
272                             BorderLayout.CENTER );
273       setSize( 300, 175 );
274       setVisible( false );
275    }
276
277    private Record getRecord()
278    {
279       Record record = new Record();
280
281       try {
282          int accountNumber = Integer.parseInt(
283                                      account.getText() );
284
285          if ( accountNumber < 1 || accountNumber > 100 ) {
286             JOptionPane.showMessageDialog( this,
287                   "Account Does Not Exist",
288                   "Error", JOptionPane.ERROR_MESSAGE );
289             return( record );
290          }
291
292          file.seek( ( accountNumber - 1 ) * Record.size() );
293          record.read( file );
294
295          if ( record.getAccount() == 0 )
296             JOptionPane.showMessageDialog( this,
297                "Account Does Not Exist",
298                "Error", JOptionPane.ERROR_MESSAGE );
299       }
300       catch ( NumberFormatException nfe ) {
301          JOptionPane.showMessageDialog( this,
302             "Invalid Account",
303             "Invalid Number Format",
304             JOptionPane.ERROR_MESSAGE );
305       }
306       catch ( IOException io ) {
307          JOptionPane.showMessageDialog( this,
308             "Error Reading File",
309             "Error", JOptionPane.ERROR_MESSAGE );
310       }
```

```
311
312         return record;
313      }
314
315      public void addRecord( Record record )
316      {
317         try {
318            int accountNumber = record.getAccount();
```

Fig. 17.14 Transaction-processing program (part 7 of 12).

```
319
320            file.seek( ( accountNumber - 1 ) * Record.size() );
321            String[] values = userInterface.getFieldValues();
322            record.write( file );
323         }
324         catch ( IOException io ) {
325            JOptionPane.showMessageDialog( this,
326               "Error Writing To File",
327               "Error", JOptionPane.ERROR_MESSAGE );
328         }
329         catch ( NumberFormatException nfe ) {
330            JOptionPane.showMessageDialog( this,
331            "Bad Balance",
332            "Invalid Number Format",
333            JOptionPane.ERROR_MESSAGE );
334         }
335      }
336   }
337
338   class NewDialog extends JInternalFrame  {
339      private RandomAccessFile file;
340      private BankUI userInterface;
341      private JButton cancel, save;
342      private JTextField account;
343
344      public NewDialog( RandomAccessFile f, ActionListener l )
345      {
346         super( "New Record" );
347
348         file = f;
349         userInterface = new BankUI();
350
351         cancel = userInterface.getDoTask();
352         cancel.setText( "Cancel" );
353         cancel.addActionListener(
354            new ActionListener() {
355               public void actionPerformed( ActionEvent e )
356               {
357                  setVisible( false );
358                  userInterface.clearFields();
359               }
360            }
361         );
362         cancel.addActionListener( l );
```

```
363
364         account = userInterface.getFields()[ BankUI.ACCOUNT ];
365         save = userInterface.getDoTask2();
366         save.setText( "Save Changes" );
367         save.addActionListener(
368              new ActionListener() {
369                 public void actionPerformed( ActionEvent e )
370                 {
371                    addRecord( getRecord() );
```

Fig. 17.14 Transaction-processing program (part 8 of 12).

```
372                    setVisible( false );
373                    userInterface.clearFields();
374                 }
375              }
376         );
377         save.addActionListener( l );
378
379         getContentPane().add( userInterface,
380                               BorderLayout.CENTER );
381         setSize( 300, 150 );
382         setVisible( false );
383      }
384
385      private Record getRecord()
386      {
387         Record record = new Record();
388
389         try {
390            int accountNumber = Integer.parseInt(
391                                        account.getText() );
392
393            if ( accountNumber < 1 || accountNumber > 100 ) {
394               JOptionPane.showMessageDialog( this,
395                  "Account Does Not Exist",
396                  "Error", JOptionPane.ERROR_MESSAGE );
397               return record;
398            }
399
400            file.seek( ( accountNumber - 1 ) * Record.size() );
401            record.read( file );
402         }
403         catch ( NumberFormatException nfe ) {
404            JOptionPane.showMessageDialog( this,
405               "Account Does Not Exist",
406               "Invalid Number Format",
407               JOptionPane.ERROR_MESSAGE );
408         }
409         catch ( IOException io ) {
410            JOptionPane.showMessageDialog( this,
411               "Error Reading File",
412               "Error", JOptionPane.ERROR_MESSAGE );
413          }
414
```

```
         return record;
      }

      public void addRecord( Record record )
      {
         int accountNumber = 0;
         String[] fields = userInterface.getFieldValues();

```

Fig. 17.14 Transaction-processing program (part 9 of 12).

```
         if ( record.getAccount() != 0 ) {
            JOptionPane.showMessageDialog( this,
               "Record Already Exists",
               "Error", JOptionPane.ERROR_MESSAGE );
            return;
         }

         // output the values to the file
         try {
            accountNumber =
               Integer.parseInt( fields[ BankUI.ACCOUNT ] );
            record.setAccount( accountNumber  );
            record.setFirstName( fields[ BankUI.FIRST ] );
            record.setLastName( fields[ BankUI.LAST ] );
            record.setBalance( Double.parseDouble(
                                 fields[ BankUI.BALANCE ] ) );
            file.seek( ( accountNumber - 1 ) * Record.size() );
            record.write( file );
         }
         catch ( NumberFormatException nfe ) {
            JOptionPane.showMessageDialog( this,
               "Invalid Balance",
               "Invalid Number Format",
               JOptionPane.ERROR_MESSAGE );
         }
         catch ( IOException io ) {
            JOptionPane.showMessageDialog( this,
               "Error Writing To File",
               "Error", JOptionPane.ERROR_MESSAGE );
         }
      }
   }

   class DeleteDialog extends JInternalFrame {
      private RandomAccessFile file;  // file for output
      private BankUI userInterface;
      private JButton cancel, delete;
      private JTextField account;

      public DeleteDialog( RandomAccessFile f, ActionListener l )
      {
         super( "Delete Record" );

         file = f;
```

```
467         userInterface = new BankUI( 1 );
468
469         cancel = userInterface.getDoTask();
470         cancel.setText( "Cancel" );
471         cancel.addActionListener(
472            new ActionListener() {
473               public void actionPerformed( ActionEvent e )
474               {
```

Fig. 17.14 Transaction-processing program (part 10 of 12).

```
475                  setVisible( false );
476               }
477            }
478         );
479         cancel.addActionListener( l );
480
481         delete = userInterface.getDoTask2();
482         delete.setText( "Delete Record" );
483         delete.addActionListener(
484            new ActionListener() {
485               public void actionPerformed( ActionEvent e )
486               {
487                  addRecord( getRecord() );
488                  setVisible( false );
489                  userInterface.clearFields();
490               }
491            }
492         );
493         delete.addActionListener( l );
494
495         account = userInterface.getFields()[ BankUI.ACCOUNT ];
496         account.addActionListener(
497            new ActionListener() {
498               public void actionPerformed( ActionEvent e )
499               {
500                  Record record = getRecord();
501               }
502            }
503         );
504         getContentPane().add( userInterface,
505                               BorderLayout.CENTER );
506
507         setSize( 300, 100 );
508         setVisible( false );
509      }
510
511      private Record getRecord()
512      {
513         Record record = new Record();
514
515         try {
516            int accountNumber = Integer.parseInt(
517                                        account.getText() );
518            if ( accountNumber < 1 || accountNumber > 100 ) {
```

```
519               JOptionPane.showMessageDialog( this,
520                  "Account Does Not Exist",
521                  "Error", JOptionPane.ERROR_MESSAGE );
522               return( record );
523            }
524
525            file.seek( ( accountNumber - 1 ) * Record.size() );
526            record.read( file );
527
```

Fig. 17.14 Transaction-processing program (part 11 of 12).

```
528            if ( record.getAccount() == 0 )
529               JOptionPane.showMessageDialog( this,
530                  "Account Does Not Exist",
531                  "Error", JOptionPane.ERROR_MESSAGE );
532         }
533         catch ( NumberFormatException nfe ) {
534            JOptionPane.showMessageDialog( this,
535               "Account Does Not Exist",
536               "Invalid Number Format",
537               JOptionPane.ERROR_MESSAGE );
538         }
539         catch ( IOException io ) {
540           JOptionPane.showMessageDialog( this,
541              "Error Reading File",
542              "Error", JOptionPane.ERROR_MESSAGE );
543         }
544
545         return record;
546      }
547
548      public void addRecord( Record record )
549      {
550         if ( record.getAccount() == 0 )
551            return;
552
553         try {
554
555            int accountNumber = record.getAccount();
556
557            file.seek( ( accountNumber - 1 ) * Record.size() );
558            record.setAccount( 0 );
559            record.write( file );
560         }
561         catch ( IOException io ) {
562            JOptionPane.showMessageDialog( this,
563               "Error Writing To File",
564               "Error", JOptionPane.ERROR_MESSAGE );
565         }
566      }
567   }
```

Fig. 17.14 Transaction-processing program (part 12 of 12).

Method	Description
`boolean canRead()`	Returns `true` if a file is readable; `false` otherwise.
`boolean canWrite()`	Returns `true` if a file is writeable; `false` otherwise.
`boolean exists()`	Returns `true` if the name specified as the argument to the `File` constructor is a file or directory in the specified path; `false` otherwise.
`boolean isFile()`	Returns `true` if the name specified as the argument to the `File` constructor is a file; `false` otherwise.
`boolean isDirectory()`	Returns `true` if the name specified as the argument to the `File` constructor is a directory; `false` otherwise.
`boolean isAbsolute()`	Returns `true` if the arguments specified to the `File` constructor indicate an absolute path to a file or directory; `false` otherwise.
`String getAbsolutePath()`	Returns a `String` with the absolute path of the file or directory.
`String getName()`	Returns a `String` with the name of the file or directory.
`String getPath()`	Returns a `String` with the path of the file or directory.
`String getParent()`	Returns a `String` with the parent directory of the file or directory—i.e., the directory in which the file or directory can be found.
`long length()`	Returns the length of the file in bytes. If the `File` object represents a directory, `0` is returned.
`long lastModified()`	Returns a platform-dependent representation of the time at which the file or directory was last modified. The value returned is only useful for comparison with other values returned by this method.
`String[] list()`	Returns an array of `String`s representing the contents of a directory.

Fig. 17.15 Some commonly used `File` methods .

```
// Fig. 17.16: FileTest.java
// Demonstrating the File class.
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;

public class FileTest extends JFrame
                      implements ActionListener {
   private JTextField enter;
   private JTextArea output;

   public FileTest()
   {
      super( "Testing class File" );

      enter = new JTextField(
         "Enter file or directory name here" );
      enter.addActionListener( this );
      output = new JTextArea();
      Container c = getContentPane();
      ScrollPane p = new ScrollPane();
      p.add( output );
      c.add( enter, BorderLayout.NORTH );
      c.add( p, BorderLayout.CENTER );

      setSize( 400, 400 );
      show();
   }

   public void actionPerformed( ActionEvent e )
   {
      File name = new File( e.getActionCommand() );

      if ( name.exists() ) {
         output.setText(
            name.getName() + " exists\n" +
            ( name.isFile() ? "is a file\n" :
                              "is not a file\n" ) +
            ( name.isDirectory() ? "is a directory\n" :
                                   "is not a directory\n" ) +
            ( name.isAbsolute() ? "is absolute path\n" :
                                  "is not absolute path\n" ) +
            "Last modified: " + name.lastModified() +
            "\nLength: " + name.length() +
            "\nPath: " + name.getPath() +
            "\nAbsolute path: " + name.getAbsolutePath() +
            "\nParent: " + name.getParent() );

      if ( name.isFile() ) {
            try {
               RandomAccessFile r =
                  new RandomAccessFile( name, "r" );
```

Fig. 17.16 Demonstrating class **File** (part 1 of 3).

```
54
55                   StringBuffer buf = new StringBuffer();
56                   String text;
57                   output.append( "\n\n" );
58
59                   while( ( text = r.readLine() ) != null )
60                      buf.append( text + "\n" );
61
62                   output.append( buf.toString() );
63                }
64                catch( IOException e2 ) {
65                   JOptionPane.showMessageDialog( this,
66                      "FILE ERROR",
67                      "FILE ERROR", JOptionPane.ERROR_MESSAGE );
68                }
69             }
70             else if ( name.isDirectory() ) {
71                String directory[] = name.list();
72
73                output.append( "\n\nDirectory contents:\n");
74
75                for ( int i = 0; i < directory.length; i++ )
76                   output.append( directory[ i ] + "\n" );
77             }
78          }
79          else {
80             JOptionPane.showMessageDialog( this,
81                e.getActionCommand() + " Does Not Exist",
82                "FILE ERROR", JOptionPane.ERROR_MESSAGE );
83          }
84       }
85
86       public static void main( String args[] )
87       {
88          FileTest app = new FileTest();
89
90          app.addWindowListener(
91             new WindowAdapter() {
92                public void windowClosing( WindowEvent e )
93                  {
94                   System.exit( 0 );
95                }
96             }
97          );
98       }
99    }
```

Fig. 17.16 Demonstrating class **`File`** (part 2 of 3).

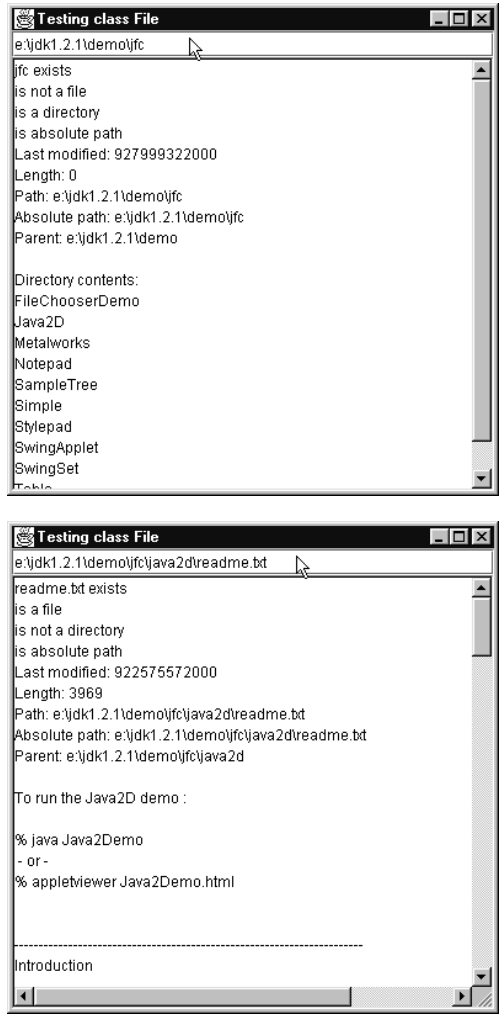

Fig. 17.16 Demonstrating class **`File`** (part 3 of 3).